Capítulo DEZESSETE
Espiando, Mapeando, Reduzindo e Coletando

Objetivos do Exame

Desenvolver código para extrair dados de um objeto usando os métodos peek() e map(), incluindo as versões primitivas do método map().

Salvar resultados em uma coleção usando o método collect e agrupar/particionar dados usando a classe Collectors.

Uso dos métodos merge() e flatMap() da API de Stream.

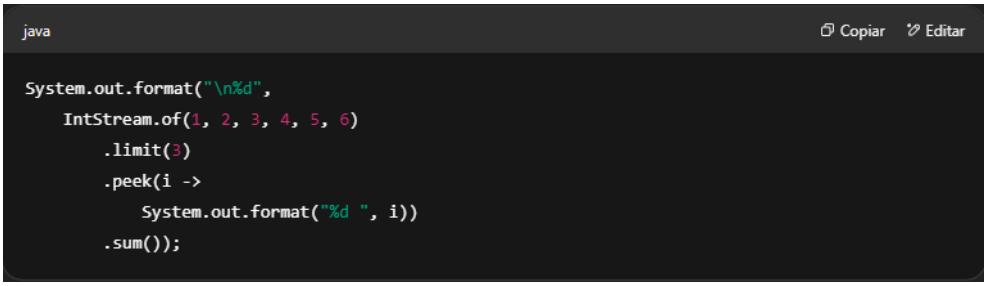
---

peek()

peek() é um método simples:

Ele apenas executa o Consumer fornecido e retorna um novo stream com os mesmos elementos do original.

Na maioria das vezes, esse método é usado com System.out.println() para fins de depuração (para ver o que há no stream):



A saída:

Observe que peek() é uma operação intermediária. No exemplo, não podemos usar algo como forEach() para imprimir os valores retornados por limit(), porque forEach() é uma operação terminal (e não poderíamos mais chamar sum()).

É importante enfatizar que peek() se destina a ver os elementos de um stream em um ponto específico do pipeline, sendo considerada uma má prática alterar o stream de qualquer forma. Se quiser fazer isso, use o método a seguir.

---

map()

map() é usado para transformar o valor ou o tipo dos elementos de um stream:

```java
<R> Stream<R> map(Function<? super T, ? extends R> mapper)

IntStream mapToInt(ToIntFunction<? super T> mapper)

LongStream mapToLong(ToLongFunction<? super T> mapper)

DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

Como pode ver, map() recebe uma Function para converter os elementos de um stream do tipo T para o tipo R, retornando um stream desse tipo R:

```java
Stream.of('a', 'b', 'c', 'd', 'e')
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

A saída:

```
97 98 99 100 101
```

Há versões para transformar em tipos primitivos, por exemplo:

```java
IntStream.of(100, 110, 120, 130, 140)
    .mapToDouble(i -> i / 3.0)
    .forEach(i -> System.out.format("%.2f ", i));
```

Irá produzir:

```
33.33 36.67 40.00 43.33 46.67
```

---

flatMap()

flatMap() é usado para “achatar” (ou combinar) os elementos de um stream em um (novo) stream:

```java
<R> Stream<R> flatMap(Function<? super T,
                      ? extends Stream<? extends R>> mapper)

DoubleStream flatMapToDouble(Function<? super T,
                      ? extends DoubleStream> mapper)

IntStream flatMapToInt(Function<? super T,
                      ? extends IntStream> mapper)

LongStream flatMapToLong(Function<? super T,
                      ? extends LongStream> mapper)
```

A partir de sua assinatura (e das assinaturas das versões primitivas), podemos ver que, em contraste com map() (que retorna um único valor), flatMap() deve retornar um Stream. Se flatMap() retornar null, um stream vazio será retornado em vez disso.

Vamos ver como isso funciona. Se temos um stream de listas de caracteres:

```java
List<Character> aToD = Arrays.asList('a', 'b', 'c', 'd');
List<Character> eToG = Arrays.asList('e', 'f', 'g');
Stream<List<Character>> stream = Stream.of(aToD, eToG);
```

E queremos converter todos os caracteres para sua representação int, não podemos mais usar map():

```java
stream.map(c -> (int)c)
```

Porque (como cada elemento do stream é passado para map) c representa um objeto do tipo List<Character>, não Character.

O que precisamos fazer é obter os elementos das listas em um único stream e então converter cada caractere para um int. Felizmente, a parte da “combinação” é exatamente o que flatMap() faz:

```java
stream
    .flatMap(l -> l.stream())
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

Então este código pode produzir:

```
97 98 99 100 101 102 103
```

Usar peek() após flatMap() pode esclarecer como os elementos são processados:

```java
stream
    .flatMap(l -> l.stream())
    .peek(System.out::print)
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

Como se pode ver na saída, o stream retornado por flatMap() é passado pelo pipeline, como se estivéssemos trabalhando com um stream de elementos únicos e não com um stream de listas de elementos:

```nginx
a97 b98 c99 d100 e101 f102 g103
```

Dessa forma, com flatMap() você pode converter um Stream<List<Object>> em um Stream<Object>. No entanto, o conceito importante é que esse método retorna um stream e não um único elemento (como map() faz).

Redução

Uma **redução** é uma operação que pega muitos elementos e os combina (ou os reduz) em um único valor ou objeto, e isso é feito aplicando uma operação várias vezes.

Alguns exemplos de reduções são somar N elementos, encontrar o maior elemento entre N números ou contar elementos.

Como no exemplo a seguir, onde usando um laço for, reduzimos um array de números à sua soma:

```java
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

Claro que fazer reduções com streams ao invés de laços tem mais benefícios, como facilitar a paralelização e melhorar a legibilidade.

A interface Stream possui dois métodos para redução:

- reduce()
- collect()

Podemos implementar reduções com ambos os métodos, mas collect() nos ajuda a implementar um tipo de redução chamado **redução mutável**, onde um recipiente (como uma Collection) é usado para acumular o resultado da operação.

---

reduce()

Este método possui três versões:

```java
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity,
         BinaryOperator<T> accumulator)

<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

Lembre-se de que um BinaryOperator<T> é equivalente a um BiFunction<T, T, T>, onde os dois argumentos e o tipo de retorno são todos do mesmo tipo.

Comecemos com a versão que recebe um argumento. Ela é equivalente a:

```java
boolean elementsFound = false;
T result = null;
for (T element : stream) {
    if (!elementsFound) {
        elementsFound = true;
        result = element;
    } else {
        result = accumulator.apply(result, element);
    }
}
return elementsFound ? Optional.of(result)
                     : Optional.empty();
```

Este código simplesmente aplica uma função a cada elemento, acumulando o resultado e retornando um Optional encapsulando esse resultado, ou um Optional vazio se não houver elementos.

Vejamos um exemplo concreto. Acabamos de ver como uma soma é uma operação de redução:

```java
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

Aqui, a operação acumuladora é:

```java
sum += n;
```

Ou:

```java
sum = sum + n;
```

O que se traduz para:

```java
OptionalInt total = IntStream.of(1, 2, 3, 4, 5, 6)
                    .reduce((sum, n) -> sum + n);
```

(Observe como a versão primitiva de stream usa a versão primitiva de Optional.)

Isso é o que acontece passo a passo:

1. Uma variável interna que acumula o resultado é inicializada com o primeiro elemento do stream (1).
2. Esse acumulador e o segundo elemento do stream (2) são passados como argumentos para o BinaryOperator representado pela expressão lambda (sum, n) -> sum + n.
3. O resultado (3) é atribuído ao acumulador.
4. O acumulador (3) e o terceiro elemento do stream (3) são passados como argumentos para o BinaryOperator.
5. O resultado (6) é atribuído ao acumulador.
6. Os passos 4 e 5 são repetidos para os próximos elementos do stream até que não haja mais elementos.

No entanto, e se você precisar ter um valor inicial? Para casos assim, temos a versão que recebe dois argumentos:

```java
T reduce(T identity, BinaryOperator<T> accumulator)
```

O primeiro argumento é o valor inicial, e ele é chamado de **identidade** porque, estritamente falando, esse valor deve ser uma identidade para a função acumuladora, ou seja, para cada valor v, accumulator.apply(identity, v) deve ser igual a v.

Esta versão de reduce() é equivalente a:

```java
T result = identity;
for (T element : stream) {
    result = accumulator.apply(result, element);
}
return result;
```

Observe que esta versão **não** retorna um objeto Optional, porque se o stream estiver vazio, o valor de identidade será retornado.

Por exemplo, o exemplo da soma pode ser reescrito como:

```java
int total = IntStream.of(1, 2, 3, 4, 5, 6)
                .reduce(0,
                        (sum, n) -> sum + n); // 21
```

Ou usando um valor inicial diferente:

```java
int total = IntStream.of(1, 2, 3, 4, 5, 6)
                .reduce(4,
                        (sum, n) -> sum + n); // 25
```

Contudo, observe que no último exemplo, o primeiro valor **não** pode ser considerado uma identidade (como no primeiro exemplo), já que, por exemplo, 4 + 1 não é igual a 4.

Isso pode causar alguns problemas ao trabalhar com **streams paralelos**, os quais revisaremos no próximo capítulo.

Agora, observe que com essas versões, você pega elementos do tipo T e retorna um valor reduzido **também do tipo T**.

Contudo, se quiser retornar um valor reduzido de um **tipo diferente**, você deve usar a versão de três argumentos de reduce():

```java
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

(Observe o uso dos tipos T e U.)

Esta versão é equivalente a:

```java
U result = identity;
for (T element : stream) {
    result = accumulator.apply(result, element);
}
return result;
```

Considere, por exemplo, que queremos obter a **soma dos comprimentos** de todas as strings de um stream. Assim, estamos recebendo strings (tipo T) e queremos um resultado inteiro (tipo U).

Nesse caso, usamos reduce() assim:

```java
int length =
    Stream.of("Parallel", "streams", "are", "great")
        .reduce(0,
                (accumInt, str) ->
                   accumInt + str.length(), // acumulador
                (accumInt1, accumInt2) ->
                   accumInt1 + accumInt2); // combinador
```

Podemos tornar mais claro adicionando os tipos dos argumentos:

```java
int length =
  Stream.of("Parallel", "streams", "are", "great")
      .reduce(0,
            (Integer accumInt, String str) ->
               accumInt + str.length(), // acumulador
            (Integer accumInt1, Integer accumInt2) ->
               accumInt1 + accumInt2); // combinador
```

Como a função acumuladora adiciona uma etapa de mapeamento (transformação) à função de acumulação, essa versão do reduce() pode ser escrita como uma combinação de map() com as outras versões do método reduce() (você talvez conheça isso como o padrão *map-reduce*):

```java
int length =
    Stream.of("Parallel", "streams", "are", "great")
        .mapToInt(s -> s.length())
        .reduce(0,
                (sum, strLength) ->
                    sum + strLength);
```

Ou simplesmente:

```java
int length = Stream.of("Parallel", "streams", "are", "great")
                .mapToInt(s -> s.length())
                .sum();
```

Porque, de fato, as operações de cálculo que aprendemos no capítulo anterior são implementadas como operações de redução nos bastidores:

- average
- count
- max
- min
- sum

Além disso, observe que se o valor retornado for do **mesmo tipo**, a função combinadora não é mais necessária (afinal, ela acaba sendo igual à função acumuladora). Assim, nesse caso, é melhor usar a versão de dois argumentos.

Recomenda-se usar o método reduce() com três argumentos quando:

- Trabalhar com **streams paralelos** (mais sobre isso no próximo capítulo).
- Ter uma única função como mapeadora e acumuladora for mais eficiente do que ter funções separadas para mapeamento e redução.

---

collect()

Este método possui duas versões:

```java
<R, A> R collect(Collector<? super T, A, R> collector)

<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner)
```

A primeira versão usa coletores predefinidos da classe Collectors, enquanto a segunda permite que você crie seus próprios coletores. Streams primitivos (como IntStream) possuem apenas essa última versão de collect().

Lembre-se de que collect() realiza uma **redução mutável** nos elementos de um stream, o que significa que ele usa um objeto mutável para acumular, como uma Collection ou um StringBuilder. Em contraste, reduce() combina dois elementos para produzir um novo e representa uma **redução imutável**.

No entanto, comecemos com a versão que recebe três argumentos, pois ela é semelhante à versão de reduce() que também recebe três argumentos.

Como pode ser visto pela sua assinatura, primeiro, ela recebe um Supplier que retorna o objeto que será usado como recipiente (acumulador) e retornado ao final.

O segundo parâmetro é uma função acumuladora, que recebe o recipiente e o elemento a ser adicionado a ele.

O terceiro parâmetro é a função de combinação (*combiner*), que mescla os resultados intermediários no resultado final (útil ao trabalhar com streams paralelos).

Esta versão de collect() é equivalente a:

```java
R result = supplier.get();
for (T element : stream) {
    accumulator.accept(result, element);
}
return result;
```

Por exemplo, se quisermos “reduzir” ou “coletar” todos os elementos de um stream em uma List, podemos fazer isso da seguinte forma:

```java
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           () -> new ArrayList<>(), // Criando o recipiente
           (l, i) -> l.add(i),      // Adicionando um elemento
           (l1, l2) -> l1.addAll(l2) // Combinando os elementos
        );
```

Podemos tornar mais claro ao adicionar os tipos dos argumentos:

```java
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           () -> new ArrayList<>(),
           (List<Integer> l, Integer i) -> l.add(i),
           (List<Integer> l1, List<Integer> l2) -> l1.addAll(l2)
        );
```

Ou também podemos usar referências de método:

```java
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           ArrayList::new,
           ArrayList::add,
           ArrayList::addAll
        );
```

Collectors

A versão anterior de collect() é útil para aprender como os coletores funcionam, mas, na prática, é melhor usar a outra versão.

Alguns coletores comuns da classe Collectors são:

Método de collect()	**Valor retornado**	**Descrição**
toList	List	Acumula os elementos em uma List.
toSet	Set	Acumula os elementos em um Set.
toCollection	Collection	Acumula os elementos em uma implementação de Collection.
toMap	Map	Acumula os elementos em um Map.
joining	String	Concatena os elementos em uma String.

Como métodos de cálculo podem ser implementados como reduções, a classe Collectors também os fornece como coletores:

Método	Valor retornado	Descrição
`averagingInt`	`Double`	Retorna a média dos elementos de entrada.
`averagingLong`	`Double`	Idem acima, para `long`.
`averagingDouble`	`Double`	Idem acima, para `double`.
`counting`	`Long`	Conta os elementos de entrada.
`maxBy`	`Optional<T>`	Retorna o maior elemento de acordo com um `Comparator`.
`minBy`	`Optional<T>`	Retorna o menor elemento de acordo com um `Comparator`.
`summingInt`	`Integer`	Retorna a soma dos elementos de entrada.
`summingLong`	`Long`	Idem acima, para `long`.
`summingDouble`	`Double`	Idem acima, para `double`.

Dessa forma, podemos reescrever o exemplo anterior:

```java
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           ArrayList::new,
           ArrayList::add,
           ArrayList::addAll
    );
```

Como:

```java
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(Collectors.toList()); // [1, 2, 3, 4, 5]
```

Como todos esses métodos são estáticos, podemos usar *imports estáticos*:

```java
import static java.util.stream.Collectors.*;

...

List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(toList()); // [1, 2, 3, 4, 5]
```

Se quisermos coletar os elementos em um Set:

```java
Set<Integer> set =
    Stream.of(1, 1, 2, 2, 2)
        .collect(toSet()); // [1, 2]
```

Se quisermos criar outra implementação de Collection:

```java
Deque<Integer> deque =
    Stream.of(1, 2, 3)
        .collect(
            toCollection(ArrayDeque::new)
    ); // [1, 2, 3]
```

Se estivermos trabalhando com streams de String, podemos unir todos os elementos em uma única String com:

```java
String s = Stream.of("a", "simple", "string")
              .collect(joining()); // "asimplestring"
```

Também podemos passar um separador:

```java
String s = Stream.of("a", "simple", "string")
             .collect(joining(" ")); // "a simple string"
```

E um prefixo e um sufixo:

```java
String s = Stream.of("a", "simple", "string")
             .collect(
                 joining(" ", "This is ", ".")
             ); // "This is a simple string."
```

---

toMap()

No caso de mapas, as coisas ficam um pouco mais complicadas, porque, dependendo das nossas necessidades, temos **três opções**.

Na primeira opção, toMap() recebe dois argumentos (não estou mostrando os tipos de retorno porque são difíceis de ler e não trazem muito valor aqui):

```java
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper)
```

Ambas as Functions recebem um elemento do stream como argumento e retornam a chave ou o valor de uma entrada do Map, por exemplo:

```java
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(
            toMap(i -> i,          // Chave
                  i -> i * 2)      // Valor
        );
```

Aqui, estamos usando o elemento (como 1) como chave, e o elemento multiplicado por dois como valor (como 2).

Também podemos escrever i -> i como Function.identity():

```java
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(
            toMap(Function.identity(), // Chave
                  i -> i * 2)          // Valor
        );
```

java.util.function.Function.identity() retorna uma função que sempre retorna seu argumento de entrada, ou seja, é equivalente a t -> t.

Mas o que acontece se mais de um elemento for mapeado para a mesma chave, como em:

```java
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(i -> i % 2, // Chave
                       i -> i));   // Valor
```

O Java não saberá o que fazer, então uma exceção será lançada:

```nginx
Exception in thread "main" java.lang.IllegalStateException: Duplicate key 1
    at java.util.stream.Collectors.lambda$throwingMerger$113(Collectors.java:133)
    at java.util.stream.Collectors$$Lambda$3/303563356.apply(Unknown Source)
    at java.util.HashMap.merge(HashMap.java:1245)
    at java.util.stream.Collectors.lambda$toMap$171(Collectors.java:1320)
```

Para esses casos, usamos a versão que recebe três argumentos:

```java
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper,
      BinaryOperator<U> mergeFunction)
```

O terceiro argumento é uma função que define o que fazer quando há uma chave duplicada. Por exemplo, podemos criar uma List para adicionar os valores:

```java
Map<Integer, List<Integer>> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(
                    i -> i % 2,
                    i -> new ArrayList<>(Arrays.asList(i)),
                    (list1, list2) -> {
                       list1.addAll(list2);
                       return list1;
                    }
                )
         );
```

Isso retornará o seguinte mapa:

```java
{0=[2, 4, 6], 1=[1, 3, 5]}
```

A terceira versão de toMap() recebe todos esses argumentos **mais um** que retorna um novo Map vazio no qual os resultados serão inseridos:

```java
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper,
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapSupplier)
```

Assim, podemos trocar a implementação padrão (HashMap) por ConcurrentHashMap, por exemplo:

```java
Map<Integer, List<Integer>> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(
                  i -> i % 2,
                  i -> new ArrayList<>(Arrays.asList(i)),
                  (list1, list2) -> {
                      list1.addAll(list2);
                      return list1;
                  },
                  ConcurrentHashMap::new
                )
        );
```

Sobre os **métodos de cálculo**, eles são fáceis de usar. Exceto por counting(), eles recebem uma Function para produzir um valor ao qual aplicar a operação, ou (no caso de maxBy e minBy) recebem um Comparator para produzir o resultado:

```java
double avg = Stream.of(1, 2, 3)
               .collect(averagingInt(i -> i * 2)); // 4.0

long count = Stream.of(1, 2, 3)
               .collect(counting()); // 3

Stream.of(1, 2, 3)
    .collect(maxBy(Comparator.naturalOrder()))
    .ifPresent(System.out::println); // 3

Integer sum = Stream.of(1, 2, 3)
    .collect(summingInt(i -> i)); // 6
```

---

groupingBy()

A classe Collectors fornece duas funções para agrupar os elementos de um stream em uma lista, em um estilo semelhante ao GROUP BY do SQL.

O primeiro método é groupingBy() e possui três versões. Esta é a primeira:

```java
groupingBy(Function<? super T, ? extends K> classifier)
```

Ela recebe uma Function que classifica elementos do tipo T, agrupa-os em uma lista e retorna o resultado em um Map onde as chaves (do tipo K) são os valores retornados pela função.

Por exemplo, se quisermos agrupar um stream de números pela faixa a que pertencem (dezenas, vintenas, etc.), podemos fazer algo assim:

```java
Map<Integer, List<Integer>> map =
   Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
      .collect(groupingBy(i -> i / 10 * 10));
```

No momento em que você compara este código com a maneira tradicional de fazer isso (com um laço for), é que percebe o poder dos streams:

```java
List<Integer> stream =
    Arrays.asList(2, 34, 54, 23, 33, 20, 59, 11, 19, 37);
Map<Integer, List<Integer>> map = new HashMap<>();

for (Integer i : stream) {
    int key = i / 10 * 10;
    List<Integer> list = map.get(key);

    if (list == null) {
        list = new ArrayList<>();
        map.put(key, list);
    }

    list.add(i);
}
```

De qualquer maneira, ambos retornarão o seguinte mapa:

```java
{0=[2], 50=[54, 59], 20=[23, 20], 10=[11, 19], 30=[34, 33, 37]}
```

A segunda versão recebe um **coletor descendente** como argumento adicional:

```java
groupingBy(Function<? super T, ? extends K> classifier,
           Collector<? super T, A, D> downstream)
```

Um **coletor descendente** é um coletor que é aplicado aos resultados de outro coletor.

Podemos usar qualquer coletor aqui, por exemplo, para contar os elementos em cada grupo do exemplo anterior:

```java
Map<Integer, Long> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
        .collect(
            groupingBy(i -> i / 10 * 10,
                       counting())
        );
```

(Observe como o tipo dos valores do Map muda para refletir o tipo retornado pelo coletor descendente, counting().)

Isso retornará o seguinte mapa:

```java
{0=1, 50=2, 20=2, 10=2, 30=3}
```

Podemos até usar outro groupingBy() para classificar os elementos em um segundo nível. Por exemplo, em vez de contar, podemos classificar ainda mais os elementos em pares ou ímpares:

```java
Map<Integer, Map<String, List<Integer>>> map =
   Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
       .collect(groupingBy(i -> i / 10 * 10,
                 groupingBy(i ->
                             i % 2 == 0 ? "EVEN" : "ODD")
                )
       );
```

Isso retornará o seguinte mapa (com um pouco de formatação):

```java
{
    0 = {EVEN=[2]},
   50 = {EVEN=[54], ODD=[59]},
   20 = {EVEN=[20], ODD=[23]},
   10 = {ODD=[11, 19]},
   30 = {EVEN=[34], ODD=[33, 37]}
}
```

A chave do mapa de nível superior é um Integer, pois o primeiro groupingBy() retorna um Integer.

O tipo dos valores do mapa de nível superior mudou (novamente) para refletir o tipo retornado pelo coletor descendente, groupingBy().

Neste caso, uma String é retornada, portanto, esse será o tipo das chaves do mapa de segundo nível, e como estamos trabalhando com um stream de Integer, os valores são do tipo List<Integer>.

Vendo a saída desses exemplos, você pode estar se perguntando: **há uma maneira de obter o resultado ordenado?**

Bem, TreeMap é a única implementação de Map que é ordenada. Felizmente, a terceira versão de groupingBy() adiciona um argumento Supplier que nos permite escolher o tipo do Map resultante:

```java
groupingBy(Function<? super T, ? extends K> classifier,
           Supplier<M> mapFactory,
           Collector<? super T, A, D> downstream)
```

Dessa forma, se passarmos uma instância de TreeMap:

```java
Map<Integer, Map<String, List<Integer>>> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
       .collect(groupingBy(i -> i / 10 * 10,
                 TreeMap::new,
                 groupingBy(i -> i % 2 == 0 ? "EVEN" : "ODD")
                )
       );
```

Isso retornará o seguinte mapa:

```java
{
    0 = {EVEN=[2]},
   10 = {ODD=[11, 19]},
   20 = {EVEN=[20], ODD=[23]},
   30 = {EVEN=[34], ODD=[33, 37]},
   50 = {EVEN=[54], ODD=[59]}
}
```

---

partitioningBy()

O segundo método para agrupamento é partitioningBy().

A diferença em relação a groupingBy() é que partitioningBy() retornará um Map com Boolean como tipo de chave, o que significa que há **apenas dois grupos**, um para true e outro para false.

Há duas versões deste método. A primeira é:

```java
partitioningBy(Predicate<? super T> predicate)
```

Ela particiona os elementos de acordo com um Predicate e os organiza em um Map<Boolean, List<T>>.

Por exemplo, se quisermos particionar um stream de números entre os que são menores que 50 e os que não são, podemos fazer assim:

```java
Map<Boolean, List<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31)
        .collect(partitioningBy(i -> i < 50));
```

Isso retornará o seguinte mapa:

```java
{false=[65, 77, 89], true=[45, 9, 12, 31]}
```

Como você pode ver, por causa do Predicate, o mapa sempre terá **dois elementos**.

E, assim como em groupingBy(), este método possui uma segunda versão que recebe um **coletor descendente**.

Por exemplo, se quisermos remover duplicatas, basta coletar os elementos em um Set desta forma:

```java
Map<Boolean, Set<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31, 12)
        .collect(
            partitioningBy(i -> i < 50,
                           toSet()
            )
        );
```

Isso produzirá o seguinte Map:

```java
{false=[65, 89, 77], true=[9, 12, 45, 31]}
```

No entanto, diferente de groupingBy(), não há uma versão que nos permita mudar o tipo do Map retornado. Mas isso não importa — você só tem duas chaves que pode obter com:

```java
Set<Integer> lessThan50 = map.get(true);
Set<Integer> moreThan50 = map.get(false);
```

Pontos-Chave (Key Points)

- peek() executa o Consumer fornecido e retorna um novo stream com os **mesmos elementos** do stream original. Na maioria das vezes, esse método é usado para fins de **depuração**.
- map() é usado para transformar o **valor ou o tipo** dos elementos de um stream por meio de uma Function fornecida.
- flatMap() é usado para "**achatar**" (ou combinar) os elementos de um stream em **um único** (novo) stream. Em contraste com map() (que retorna um único valor), flatMap() **deve retornar um Stream**.
- Uma **redução** é uma operação que pega muitos elementos e os **combina** (ou reduz) em um único valor ou objeto.
- reduce() realiza uma redução nos elementos de um stream usando uma função acumuladora, um valor de identidade opcional e também uma função combinadora opcional.
- collect() implementa um tipo de redução chamado **redução mutável**, em que um recipiente (como uma Collection) é usado para acumular o resultado da operação.
- A classe Collectors fornece métodos estáticos como toList() e toMap() para criar uma coleção ou um mapa a partir de um stream, além de métodos de cálculo como averagingInt().
- Collectors.groupingBy() agrupa os elementos de um stream usando uma Function fornecida como classificador. Ele também pode receber um **coletor descendente** para criar outro nível de classificação.
- Você também pode **agrupar** (ou particionar) os elementos em um stream com base em uma condição (Predicate) usando o método Collectors.partitioningBy().

Autoavaliação (Self Test)

1. Dado:

```java
public class Question_17_1 {
    public static void main(String[] args) {
        Map<Boolean, List<Integer>> map =
           Stream.of(1, 2, 3, 4)
              .collect(partitioningBy(i -> i < 5));
        System.out.println(map);
    }
}
```

Qual é o resultado?

A. {true=[1, 2, 3, 4]}
B. {false=[], true=[1, 2, 3, 4]}
C. {false=[1, 2, 3, 4]}
D. {false=[1, 2, 3, 4], true=[]}

---

2. Dado:

```java
groupingBy(i -> i % 3, toList())
```

Qual das seguintes opções é equivalente?

A. partitioningBy(i -> i % 3 == 0, toList())
B. partitioningBy(i -> i % 3, toList())
C. groupingBy(i -> i % 3 == 0)
D. groupingBy(i -> i % 3)

---

3. Dado:

```java
public class Question_17_3 {
    public static void main(String[] args) {
        Stream.of("aaaaa", "bbbb", "ccc")
           .map(s -> s.split(""))
           .limit(1)
           .forEach(System.out::print);
    }
}
```

Qual é o resultado?

A. aaaaa
B. abc
C. a
D. Nenhuma das anteriores

---

4. Dado:

```java
public class Question_17_4 {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("a", "b", "c")
                .flatMap(s -> Stream.of(s, s, s))
                .collect(Collectors.toList())
        );
    }
}
```

Qual é o resultado?

A. [a, a, a, b, b, b, c, c, c]
B. [a, a, a]
C. [a, b, c]
D. Falha de compilação

---

5. Qual das opções a seguir é a maneira correta de implementar OptionalInt min() com uma operação de redução?

A. reduce((a, b) -> a > b)
B. reduce(Math::min)
C. reduce(Integer.MIN_VALUE, Math:min)
D. collect(Collectors.minBy())

---

6. Qual das seguintes é uma sobrecarga correta do método reduce()?

A.

```java
T reduce(BinaryOperator<T> accumulator)
```

B.

```java
Optional<T> reduce(T identity,
          BinaryOperator<T> accumulator)
```

C.

```java
<U> U reduce(BinaryOperator<T> accumulator,
             BinaryOperator<U> combiner)
```

D.

```java
<U> U reduce(U identity,
          BiFunction<U, ? super T, U> accumulator,
          BinaryOperator<U> combiner)
```

---

7. Dado:

```java
public class Question_17_7 {
    public static void main(String[] args) {
        Map<Integer, Map<String, List<Integer>>> map =
            Stream.of(56, 54, 1, 31, 98, 98, 16)
                .collect(groupingBy(
                            i -> i % 10,
                            TreeMap::new,
                            partitioningBy(i -> i > 5)
                        )
                );
        System.out.println(map);
    }
}
```

Qual é o resultado?

A.

```java
{
   6={false=[], true=[56, 16]},
   4={false=[], true=[54]},
   1={false=[1], true=[31]},
   8={false=[], true=[98]}
}
```

B.

```java
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]},
   6={false=[], true=[56, 16]},
   8={false=[], true=[98]}
}
```

C.

```java
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]},
   6={false=[], true=[56, 16]},
   8={false=[], true=[98, 98]}
}
```

D.

```java
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]}
}
```